Isogeny School 2020: Constant-time implementations of isogeny schemes

Michael Meyer

RheinMain University of Applied Sciences, Wiesbaden, Germany
`michael@random-oracles.org`

19th September 2021

These notes discuss constant-time implementations of isogeny-based schemes. Specifically, several papers have investigated how to implement CSIDH efficiently in constant time [8, 10, 4, 5, 1]. In contrast, for SIDH this is rather straightforward, since the isogeny computations have a constant running time by design. We will thus focus on CSIDH.

The style of these notes is rather informal, but references to more formal and precise literature are included. The discussion will focus on algorithmic aspects of CSIDH, and thus stay on a high level. Low-level details such as constant-time C implementations of field arithmetic or other techniques we will use are beyond the scope of these notes (see, e.g., [11]).

1 Constant-time implementations

As a warm-up, we postpone our entry to isogeny-land, and discuss general properties of constant-time implementations, and a simple example.

First, we need to define what we mean by "constant time". We'll discuss this along the example of exponentiation in a discrete logarithm group G (writing the group multiplicatively). In particular, we usually have a public input $g \in G$ of large prime order p, and a private key a such that $1 \leq a < p$. Our goal is to compute g^a without leaking information on a.

A reasonable definition could say that a constant-time implementation of this exponentiation should have a constant running time, independent of the secret a. Although this definition might hold in some situations, this is not what cryptographers usually mean by "constant time".

Definition. An implementation is said to be constant-time if its running time is independent of the secret input.

See [1, §2.4] for a more detailed definition and explanation.

Note that this definition implies that any function inside the implementation must perform its operations independent of the input, if the input is in some way derived from the secret. In particular, the implementation is not allowed to contain conditional branches that depend on the secret, such as `if`-statements whose conditions are derived from the secret input. Furthermore, we have to avoid reading from or writing to memory locations that depend on the secret, since this would potentially allow for cache-timing attacks. An example for this are array indices that depend on the secret, such as in lookup tables.

Now let's see the theory in action, through the example of exponentiation. A simple and efficient way to compute g^a for any $1 \leq a < p$ is the square-and-multiply method.[1] Throughout this example we assume that squarings and multiplications in the group G are constant-time, i.e., run independent of the input values. The perhaps simplest implementation of square-and-multiply could use the following algorithm. In the following, we write $a_0, \ldots, a_{k-1}$ to denote the bit representation of a k-bit number a, i.e., we have $a = \sum_{i=0}^{k-1}(a_i \cdot 2^i)$ with $a_i \in \{0, 1\}$ and $a_{k-1} = 1$.

Algorithm 1: Square-and-multiply.

```
  Input  : $g, a$
  Output: $g^a$
1 $x \leftarrow 1$
2 foreach $i \in \{k-1, k-2, \ldots, 0\}$ do
3     $x \leftarrow x \cdot x$
4     if $a_i = 1$ then
5         $x \leftarrow x \cdot g$
6 return $x$
```

This algorithm is very simple and efficient, but quite obviously, it has flaws in terms of constant-time requirements when the exponent a is secret:

- The secret a is chosen such that $1 \leq a < p$, so in general the bitlength may vary between different choices of a. This means that the number of steps in Algorithm 1 may vary too.
- Even if we have distinct secret exponents a and a' of the same bitlength, the number of multiplications required in Line 5 depends on the number of set bits of a resp. a'. Thus, the running time depends on the number of set bits of the exponent.

There are various options to turn Algorithm 1 into a constant-time algorithm. A simple approach is the following.

- Let $k = \lceil \log_2 p \rceil$ be the bitlength of p. Then we represent every valid exponent a as a k-bit number by padding with leading 0s if necessary. This means that if some a has $k' < k$ bits, we set $a_{k-1} = a_{k-2} = \cdots = a_{k'} = 0$. This obviously does not change the result of Algorithm 1, but fixes the number of steps to be k, independent of the secret a.
- Algorithm 1 uses k squarings and as many multiplications as a has set bits. A simple way to make the number of multiplications independent of a is to compute a multiplication in every step, but throw away the result if the corresponding $a_i = 0$. These additional multiplications are *dummy operations*, which let us fix the total number of operations to k squarings plus k multiplications, independent of a.

[1] In groups that are written additively, such as points on elliptic curves, the analogous method is double-and-add.

The resulting algorithm could then looks as in Algorithm 2, where we assume that k is fixed as explained above, and a may have leading zero bits if required.

Algorithm 2: Square-and-multiply with a fixed number of operations.

```
Input  : $g, a$
Output: $g^a$
1 $x \leftarrow 1$
2 foreach $i \in \{k-1, k-2, \ldots, 0\}$ do
3     $x \leftarrow x \cdot x$
4     if $a_i = 1$ then
5         $x \leftarrow x \cdot g$
6     if $a_i = 0$ then
7         $x' \leftarrow x \cdot g$
8 return $x$
```

If the multiplication and squaring operations themselves are constant-time, then Algorithm 2 appears to be constant-time too: There's a fixed number of constant-time operations. However, we face one more problem here. Independent of the bits a_i, we compute $x \cdot g$ at each step. The crucial point is that we either write the result to x if $a_i = 1$ or to x' if $a_i = 0$. This means that we write to a memory location that depends on the secret, and therefore Algorithm 2 is not constant-time.

In order to fix this sort of problem, implementers usually use constant-time low-level functions. In our case, we can use a constant-time conditional swap function `cswap`(x, x', b), which leaves x and x' unchanged if the decision bit $b = 0$, and swaps the values of x and x' if $b = 1$. For details on how to implement such function in constant-time, see [11]. Instead of using an `if`-branch and writing results to the according variable, we can now do the following.

Algorithm 3: Constant-time square-and-multiply.

```
Input  : $g, a$
Output: $g^a$
1 $x \leftarrow 1$
2 foreach $i \in \{k-1, k-2, \ldots, 0\}$ do
3     $x \leftarrow x \cdot x$
4     $x' \leftarrow x \cdot g$
5     $x, x' \leftarrow$ cswap$(x, x', a_i)$
6 return $x$
```

Other side-channel attacks. Constant-time implementations only prevent certain kinds of side channel attacks, where the attacker only observes the execution of an algorithm, and tries to gain information e.g. from timings. However, this does not prevent other side-channel attacks such as power analysis, or active side-channel attacks such as fault injection attacks. These kinds of attacks are beyond the scope of these notes. We refer to [7] for more information.

2 CSIDH computations

In this section we briefly recall how a CSIDH [3] group action evaluation is computed. For more details, see Tanja Lange's notes from week 3 of the summer school [6].

We have a prime of the form $p = 4 \cdot \ell_1 \cdot \dots \cdot \ell_n - 1$, where the ℓ_i are small distinct odd primes and $\ell_1 < \ell_2 < \dots < \ell_n$. We work with supersingular elliptic curves over $\mathbb{F}_p$ in Montgomery form $E_A : y^2 = x^3 + Ax^2 + x$, represented by their A-coefficients. The supersingularity of the involved curves implies that their group order is $p + 1 = 4 \cdot \ell_1 \cdot \dots \cdot \ell_n$. Thus, $\mathbb{F}_p$-rational points (with coordinates $x, y \in \mathbb{F}_p$) of orders $\ell_1, \dots, \ell_n$ exist on these curves.

For any starting curve E_A, such a point K of order ℓ_i generates a subgroup of order ℓ_i, which in turn gives us a unique (up to isomorphisms) isogeny $\varphi_i : E_A \to E_{A'} = E_A/\langle K \rangle$ with kernel $\langle K \rangle$. For this computation, we usually write $\mathfrak{l}_i * E_A$. For finding a point K of suitable order ℓ_i, we can sample a random point $P \in E_A$, whose order then divides the group order $p+1$, and compute $K = [(p+1)/\ell_i]P$. If $K \neq \infty$, we have found a suitable point, otherwise, we repeat this procedure.

On the other hand, we can also compute an "inverse" of such an isogeny, in the sense that $\mathfrak{l}_i^{-1} * (\mathfrak{l}_i * E_A) = \mathfrak{l}_i * (\mathfrak{l}_i^{-1} * E_A) = E_A$. An isogeny corresponding to $\mathfrak{l}_i^{-1}$ can be computed by using a kernel $\langle K \rangle$ of order ℓ_i, where $K = (x, y)$ with $x \in \mathbb{F}_p$ and $y \in \mathbb{F}_{p^2} \backslash \mathbb{F}_p$. Finding a suitable point works analogously to the previous case, with the modification of sampling an initial point with coordinates $x \in \mathbb{F}_p$ and $y \in \mathbb{F}_{p^2} \backslash \mathbb{F}_p$. In practice, we can sample a random x-coordinate, and check whether the corresponding y-coordinates are defined over $\mathbb{F}_p$ or $\mathbb{F}_{p^2} \backslash \mathbb{F}_p$.

CSIDH key space. In the CSIDH setting, we can efficiently compute isogenies corresponding to $\mathfrak{l}_1, \dots \mathfrak{l}_n$ (and their inverses). We can also apply each $\mathfrak{l}_i$ multiple times, so we can efficiently compute the action of $\prod_{i=1}^n \mathfrak{l}_i^{e_i}$ for exponents e_i of small absolute value.

In total, there are about $\sqrt{p}$ curves that we can reach from a starting curve E_A via isogenies. Thus, in CSIDH we sample the exponents from certain ranges, i.e., we sample $e_i \in [-m_i, m_i]$, such that we can obtain about $\sqrt{p}$ different exponent vectors. Under the heuristic that there are not many collisions among these vectors, i.e., different vectors that produce the same output curve, this means that we can reach almost all valid curves with this key space. For instance, the CSIDH-512 parameter set has $\sqrt{p} \approx 2^{256}$, $n = 74$, and $m_i = 5$ for all i. This means that the key space has size $11^{74} \approx \sqrt{p}$.

It is important to note that the action of the different $\mathfrak{l}_i$ is commutative. In particular, if we compute a series of isogenies, say $[\mathfrak{l}_i \mathfrak{l}_j \mathfrak{l}_k] * E_A$, we can compute the respective isogenies in any order, and always obtain the same resulting curve $E_{A'}$. The commutativity also allows us to define a Diffie–Hellman-style key exchange in a straightforward way.

Efficient computation. The process described above for computing isogenies is rather inefficient in our setting. Instead of sampling at least one point per isogeny, we can combine isogeny computations from only one sampled point. As an example, suppose we want to compute an ℓ_i- and ℓ_j-isogeny, and that $e_i > 0$ and $e_j > 0$. Then we can sample a point P_0 on the starting curve with suitable y-coordinate ($y \in \mathbb{F}_p$ in this case), and first compute $P \leftarrow [(p+1)/(\ell_i \ell_j)]P_0$. Then the

Algorithm 4: Evaluating the class group action.

```
Input  : $A \in \mathbb{F}_p$ and a list of integers $(e_1, \ldots, e_n)$.
Output: $A'$ such that $[\mathfrak{l}_1^{e_1} \cdot \cdots \cdot \mathfrak{l}_n^{e_n}] * E_A = E_{A'}$.
1  while some $e_i \neq 0$ do
2      Sample a random $x \in \mathbb{F}_p$.
3      Set $s \leftarrow +1$ if $x^3 + Ax^2 + x$ is a square in $\mathbb{F}_p$, else $s \leftarrow -1$.
4      Let $S = \{i \mid \mathrm{sign}(e_i) = s\}$.
5      if $S = \emptyset$ then
6          Go to line 2.
7      $P = (x : 1), k \leftarrow \prod_{i \in S} \ell_i, P \leftarrow [(p+1)/k]P$.
8      foreach $i \in S$ do
9          $K \leftarrow [k/\ell_i]P$
10         if $K \neq \infty$ then
11             Compute a degree-$\ell_i$ isogeny $\varphi : E_A \to E_{A'}$ with $\ker(\varphi) = \langle K \rangle$:
12             $A \leftarrow A', P \leftarrow \varphi(P), k \leftarrow k/\ell_i, e_i \leftarrow e_i - s$.
```

order of P divides $\ell_i \ell_j$. We compute $K \leftarrow [\ell_j]P$, and if $K \neq \infty$, it has order ℓ_i and can be used to compute an ℓ_i-isogeny φ_i. We then push P through this isogeny by computing $P \leftarrow \varphi_i(P)$, which means that the order of P loses the factor ℓ_i.[2] Then we set $K \leftarrow P$ and either have $K = \infty$, or K has order ℓ_j, in which case we compute an ℓ_j-isogeny φ_j.

Similarly, we can combine the computation of an arbitrary set of isogenies of different degrees, as long as the corresponding e_i share the same sign.

An efficient algorithm to compute the CSIDH action is shown in Algorithm 4.

3 Constant-time CSIDH

The goal of this section is to apply the ideas of Section 1 to CSIDH as in Algorithm 4. Again we assume that lower-level functions such as field multiplications are constant-time. This means that we can also assume that if given a kernel generator K, the computation of the isogeny with kernel $\langle K \rangle$ does not leak any information except for its degree.

Similar to the simple square-and-multiply approach from Algorithm 1, Algorithm 4 is obviously not constant-time.

Number of isogenies. Analogous to the variable number of multiplications in Algorithm 1, Algorithm 4 computes a variable number of isogenies. In particular, for the private key vector $(e_1, e_2, \ldots, e_n)$, we compute $|e_1|$ isogenies of degree ℓ_1, $|e_2|$ isogenies of degree ℓ_2, etc. In the case of square-and-multiply, we solved this by adding dummy multiplications, such that each step contains a multiplication, independent of the secret (see Algorithm 2).

[2]Note that the order of a point $P = (x, y)$ with $x, y \in \mathbb{F}_p$ stays unchanged when being pushed through an isogeny corresponding to an $\mathfrak{l}_i^{-1}$. The same is true for a point $P = (x, y)$ with $x \in \mathbb{F}_p$ and $y \in \mathbb{F}_{p^2} \backslash \mathbb{F}_p$ and an isogeny corresponding to an $\mathfrak{l}_i$.

In Algorithm 4, we can apply the same idea: In addition to the actually required isogenies, determined by the private key, we compute *dummy isogenies*, whose results are discarded.[3] However, for each degree ℓ_i, the cost for computing an ℓ_i-isogeny depends on the size of ℓ_i. Thus, it is not only the total number of isogenies that we have to fix, but the number of isogenies per degree ℓ_i. We can do this by computing m_i isogenies per degree ℓ_i, where $|e_i|$ of them are actual isogenies, and $m_i - |e_i|$ are dummy isogenies.

Note that again we have to make sure not to use secret-dependent `if`-branches or memory accesses, e.g. via the usage of functions such as `cswap`.

Point rejections and multiplications. In order to compute an ℓ_i-isogeny, we have to sample a point and hope for its order to contain the factor ℓ_i. Otherwise, the check in Line 10 of Algorithm 4 fails. This means that the running time of this algorithm cannot be constant. In particular, the probability for this check to succeed is $1 - 1/\ell_i$ for all ℓ_i, and therefore the number of necessary attempts relies on our luck with sampling points. Nevertheless, we can use this sampling method in a constant-time implementation, since we don't require the running time to be constant, but independent of secrets. What we want to keep secret, is how many of the m_i isogenies of degree ℓ_i are real isogenies, and how many are dummy isogenies. Thus, we have to enforce the condition of Line 10 even in the case of dummy isogenies, such that the number of necessary attempts for ℓ_i-isogenies only depends on m_i and randomness, but not on the secret e_i.

Furthermore, the multiplicative effort for computing a point K for the check in Line 10 depends on the respective ℓ_i. However, we don't have to keep secret which degree the current isogeny has, but only if it is an actually required isogeny or a dummy isogeny. Thus, we don't have to use constant-time algorithms for the involved scalar multiplications.

Sign distribution. Besides the number of isogenies, the sign distribution of the private key elements e_i must be kept secret. However, Algorithm 4 has a different behavior for different sign distributions. Consider the keys $e = (e_1, \dots, e_n)$ with $e_i > 0$ for all i, and $e' = (e_1, -e_2, \dots, e_{n-1}, -e_n)$. When running the group action with the key e, in the first round the outer multiplication (Line 7) is a multiplication by 4, while the first inner multiplication (Line 9) starts with factor $(p+1)/(4\ell_i)$, and decreases over the following loops. For e', the situation is different. The outer multiplication has a factor of roughly $\sqrt{p}$ in the first round, while the inner multiplications start with factors of size roughly $\sqrt{p}$, too.

In order to satisfy the constant-time requirements, we have to avoid this secret-dependent behavior. One possibility would be to use constant-time scalar multiplications, which would lead to a massive loss of performance. A more efficient and much simpler way to achieve this is a small adaption of the key sampling. Instead of sampling the key elements e_i from $[-m_i, m_i]$, we sample them from $[0, 2m_i]$. This means that we still have the same key space size, but now all key elements are non-negative, and therefore there is nothing that could leak about this sign distribution. On the downside, we now have to compute twice as many isogenies as before, and thus our constant-time algorithm needs roughly twice as much time.

Algorithm 5 summarizes the mentioned ideas in a constant-time algorithm for CSIDH.

[3] Note that we need an extra scalar multiplication $P \leftarrow [\ell_i]P$ in the dummy case, since P doesn't lose this factor of its order by being pushed through an isogeny. In order to reach the constant-time requirements, these multiplications must also be computed after actual isogeny computations. See [8] for a way to merge this extra multiplication with dummy isogenies.

Algorithm 5: Constant-time evaluation of the class group action in CSIDH.

```
Input  : $A \in \mathbb{F}_p$ and a list of integers $(e_1, \dots, e_n)$ with $e_i \in [0, 2m_i]$ for all $i \leq n$.
Output: $A' \in \mathbb{F}_p$, the curve parameter of the resulting curve $E_{A'}$.
1  Initialize $k = 4$, $e = (e_1, \dots, e_n)$ and $f = (f_1, \dots, f_n)$, where $f_i = 2m_i - e_i$.
2  while some $e_i \neq 0$ or $f_i \neq 0$ do
3      Sample random values $x \in \mathbb{F}_p$ until we have some $x$ where $x^3 + ax^2 + x$ is a square in $\mathbb{F}_p$.
4      Set $P = (x : 1)$, $P \leftarrow [k]P$, $S = \{i \mid e_i \neq 0$ or $f_i \neq 0\}$.
5      foreach $i \in S$ do
6          Let $m = \prod_{j \in S, j > i} \ell_j$.
7          Set $K \leftarrow [m]P$.
8          if $K \neq \infty$ then
9              Set $b = 1$ if $e_i > 0$, and $b = 0$ otherwise.
10             Compute a degree-$\ell_i$ isogeny $\varphi : E_A \to E_{A'}$ with $\ker(\varphi) = \langle K \rangle$.
11             Compute $P' \leftarrow \varphi(P)$.
12             cswap$(A, A', b)$
13             cswap$(P, P', b)$
14             $e_i \leftarrow e_i - b$
15             $f_i \leftarrow f_i - (1 - b)$
16             $P \leftarrow [\ell_i]P$
17             if $e_i = 0$ and $f_i = 0$ then
18                 Set $k \leftarrow k \cdot \ell_i$.
```

A more efficient method lets us go back to sampling key elements from $[-m_i, m_i]$. The main idea is to always keep two points instead of only one as in Algorithm 4 and Algorithm 5. In particular, we sample points $P_+ = (x, y)$ with $x, y \in \mathbb{F}_p$ and $P_- = (x', y')$ with $x' \in \mathbb{F}_p$, $y' \in \mathbb{F}_{p^2} \backslash \mathbb{F}_p$. We then keep both points during the loops in the algorithm, and secretly choose the suitable point for computing a potential kernel generator for each isogeny. The details are left as an exercise.

Exercise 1. Write an algorithm for constant-time CSIDH using the 2-point method from above. Is your algorithm more efficient than Algorithm 5?

Exercise 2. Write an algorithm for constant-time CSIDH that does not use dummy isogenies. Is your algorithm more efficient than Algorithm 5?

Hint: Section 2 states a property that could be very helpful here.

Exercise 3. Although being constant-time, the running time of Algorithm 5 depends on randomness. How do we have to vary the parameters of CSIDH-512 ($n = 74$, $m_i = 5$ for all i) in order to allow for a randomness-free constant-time implementation with equally large key space of roughly 2^{256}?

4 CTIDH

[1] recently proposed a more efficient way for constant-time CSIDH. The main idea of CTIDH is to use a different key space, and an adapted algorithm to evaluate the respective group action in constant-time.

CTIDH key space. Recall that CSIDH usually samples key elements $e_i \in [-m_i, m_i]$ for certain bounds m_i. Instead of this sampling method, [9] showed that it is more efficient to fix a suitable upper bound B and sample e_i such that $\sum |e_i| \leq B$. However, a constant-time implementation of this idea then would have to hide which degree each computed isogeny has, which amounts to a massive overhead.

CTIDH uses the idea of bounds for 1-norms too, but in a different setting. We organize the prime degrees $\ell_1, \ldots, \ell_n$ in *batches*, for instance $(\ell_1, \ell_2, \ell_3), (\ell_4, \ldots, \ell_{10}), \ldots (\ell_{n-2}, \ell_{n-1}), (\ell_n)$. For each of these batches, we fix an upper bound B_i for the 1-norm of the respective key elements. In our example this would mean that we e.g. require $|e_1| + |e_2| + |e_3| \leq B_1$ when sampling a key.

Now the question is what we gain from this setting. As a toy example, assume that we have 6 prime degrees $\ell_1, \ldots, \ell_6$ available. In the usual CSIDH setting, we could sample $e_i \in [0, 1]$, which means that we have to compute 6 isogenies in a constant-time implementation, and obtain a key space of size $2^6 = 64$. In CTIDH, when instantiating two batches of size 3, sampling $e_i \in [0, 3]$ with the conditions $e_1 + e_2 + e_3 \leq 3$ and $e_4 + e_5 + e_6 \leq 3$, we get $20^2 = 400$ possible keys, while again having to compute 6 isogenies in a constant-time implementation. Thus, this setting allows us to achieve the same key space sizes with fewer isogenies to compute. In practice, the previously fastest constant-time implementation of CSIDH-512 requires a total of 438 isogenies, while the best known CTIDH parameters only need 208 isogenies for the same security level. In exchange, it is more difficult to get the computations done in constant-time.

Exercise 4. Assume that we have a batch of k prime degrees $(\ell_1, \ldots, \ell_k)$, and that we can sample the respective key elements $e_i \in [-B, B]$ for a given bound B>0, under the condition that $\sum_{i=1}^{k} |e_i| \leq B$. How many different key batch vectors $(e_1, \ldots, e_k)$ does this sampling method admit.

The CTIDH algorithm. As in the CSIDH constant-time algorithms, CTIDH has to keep the number of dummy vs. non-dummy isogenies secret, as well as the sign distribution of the key elements. However, for a batch $(\ell_1, \ldots, \ell_k)$, we only impose the condition $\sum |e_i| \leq B$ for a bound B, which means that the number of ℓ_i-isogenies for each i varies with the key. Using the usual isogeny formulas, the running time of an ℓ_i-isogeny computation depends on ℓ_i. This means that a straightforward implementation of isogenies would result in a variable running time, depending of how the key elements are distributed among the batch. In particular, we have to make sure that key batches such as $(B, 0, \ldots, 0)$ and $(0, \ldots, 0, B)$ do not lead to varying running times of the algorithm.

There are three problems that we have to solve:

- Computing an ℓ_i-isogeny with Vélu-style formulas essentially amounts to evaluating polynomials of the form

$$h_{\ell_i}(x) = \prod_{j=1}^{(\ell_i-1)/2} (x - x([j]P))$$

for an input point P of order ℓ_i, where $x(P)$ denotes the x-coordinate of P. In CTIDH, we have to make sure that all B_i isogenies for the i-th batch have the same running time, independent of the isogeny degrees. We achieve this by using the *Matryoshka structure* of the isogeny formulas. Given prime degrees ℓ_i, ℓ_j with $\ell_i < \ell_j$, computing $h_{\ell_j}(x)$ in an ℓ_j-isogeny contains all the necessary operations for computing $h_{\ell_i}(x)$, plus a few extra steps. Thus, if we compute an ℓ_i-isogeny, we can add these few extra steps as dummy operations, and the running time is the same as for an ℓ_j-isogeny. Thus, in CTIDH we can compute each isogeny of a batch at the cost of the maximal degree of this batch.

- After sampling a point P, the scalar multiplications for computing a potential kernel generator point K of order ℓ_i depends on the size of ℓ_i. However, in CTIDH we have to make sure that this takes the same time for any prime degree in a batch. We leave this as an exercise.

- The point rejection probability for a point of order ℓ_i is $1/\ell_i$, so again depends on the secret choice of an ℓ_i from a batch $(\ell_1, \ldots, \ell_k)$. In particular, for the smallest prime factor of a batch, in this case ℓ_1, this probability is the highest. Therefore, we have to make sure that the rejection probability is the same for all degrees within a batch. We do this by performing an additional coin toss with success probability $(\ell_i \cdot (\ell_1 - 1))/(\ell_1 \cdot (\ell_i - 1))$ for the respective ℓ_i. Then we only continue with computing an ℓ_i-isogeny if the respective point $K \neq \infty$ *and* the coin toss succeeds. This ensures that for all degrees in the batch, the rejection probability is $1/\ell_1$, independent of ℓ_i.

Another important topic is the choice of parameters. This is more complicated, since in addition to the CSIDH parameters, we have to define the CTIDH batches. It is not known how to optimally instantiate CTIDH, but several ideas for finding good parameters are explained in [1]. In total, implementing all these techniques in constant-time with the best known CTIDH parameters results in a speedup of almost 50% over the previously fastest constant-time implementations of CSIDH for different security levels. See [1] for more details.

Exercise 5. Show that the Vélusqrt isogeny formulas [2] have a similar Matryoshka structure, and can therefore be used in CTIDH.

Exercise 6. Given the CTIDH batch $(\ell_1, \ldots, \ell_k)$ and a point P of full order $p+1$, write an algorithm that computes a point of order ℓ_i from P, without leaking which $\ell_i \in \{\ell_1, \ldots, \ell_k\}$ was used.

References

[1] Gustavo Banegas, Daniel J. Bernstein, Fabio Campos, Tung Chou, Tanja Lange, Michael Meyer, Benjamin Smith, and Jana Sotáková. CTIDH: faster constant-time CSIDH. Cryptology ePrint Archive, Report 2021/633, 2021. `https://ia.cr/2021/633`.

[2] Daniel J. Bernstein, Luca De Feo, Antonin Leroux, and Benjamin Smith. Faster computation of isogenies of large prime degree. ANTS-XIV, 2020. `https://eprint.iacr.org/2020/341`.

[3] Wouter Castryck, Tanja Lange, Chloe Martindale, Lorenz Panny, and Joost Renes. CSIDH: An efficient post-quantum commutative group action. In T. Peyrin and S. D. Galbraith, editors, *Advances in Cryptology – ASIACRYPT 2018*, pages 395–427. Springer, 2018.

[4] Daniel Cervantes-Vázquez, Mathilde Chenu, Jesús-Javier Chi-Domínguez, Luca De Feo, Francisco Rodríguez-Henríquez, and Benjamin Smith. Stronger and Faster Side-Channel Protections for CSIDH. In P. Schwabe and N. Thériault, editors, *Progress in Cryptology – LATINCRYPT 2019*, pages 173–193. Springer, 2019.

[5] Jesús-Javier Chi-Domínguez and Francisco Rodríguez-Henríquez. Optimal strategies for CSIDH. Cryptology ePrint Archive, Report 2020/417, 2020. https://eprint.iacr.org/2020/417.

[6] Tanja Lange. (C)SIDH – Isogeny school week 3. Summer school on real-world crypto and privacy, Šibenik, Croatia, 2021. https://www.hyperelliptic.org/tanja/teaching/isogeny-school21/.

[7] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power analysis attacks: Revealing the secrets of smart cards*. Springer Science & Business Media, 2008.

[8] Michael Meyer, Fabio Campos, and Steffen Reith. On Lions and Elligators: An efficient constant-time implementation of CSIDH. In J. Ding and R. Steinwandt, editors, *Post-Quantum Cryptography – 10th International Conference, PQCrypto 2019*, pages 307–325. Springer, 2019.

[9] Kohei Nakagawa, Hiroshi Onuki, Atsushi Takayasu, and Tsuyoshi Takagi. L_1-Norm Ball for CSIDH: Optimal Strategy for Choosing the Secret Key Space. Cryptology ePrint Archive, Report 2020/181, 2020. https://ia.cr/2020/181.

[10] Hiroshi Onuki, Yusuke Aikawa, Tsutomu Yamazaki, and Tsuyoshi Takagi. (Short Paper) A Faster Constant-Time Algorithm of CSIDH Keeping Two Points. In N. Attrapadung and T. Yagi, editors, *Advances in Information and Computer Security – 14th International Workshop on Security, IWSEC 2019*, pages 23–33. Springer, 2019.

[11] Peter Schwabe. Timing Attacks and Countermeasures. Summer school on real-world crypto and privacy, Šibenik, Croatia, 2016. https://summerschool-croatia.cs.ru.nl/2016/slides/PeterSchwabe.pdf.